HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET Estimates in Multicore Real-Time Systems

Pedro Benedicte[1]
Barcelona Supercomputing Center and Universitat Politècnica de Catalunya, Barcelona, Spain
pbenedic@bsc.es
https://orcid.org/0000-0003-1670-7783

Carles Hernandez[2]
Barcelona Supercomputing Center, Barcelona, Spain
carles.hernandez@bsc.es
https://orcid.org/0000-0001-5393-3195

Jaume Abella[3]
Barcelona Supercomputing Center, Barcelona, Spain
jaume.abella@bsc.es
https://orcid.org/0000-0001-7951-4028

Francisco J. Cazorla[4]
Barcelona Supercomputing Center and IIIA-CSIC, Barcelona, Spain
francisco.cazorla@bsc.es
https://orcid.org/0000-0002-3344-376X

Abstract

High-performance processors have deployed multilevel cache (MLC) systems for decades. In the embedded real-time market, the use of MLC is also on the rise, with processors for future systems in space, railway, avionics and automotive already featuring two or more cache levels. One of the most critical elements for MLC is the write policy that not only affects several key metrics such as performance, WCET estimates, energy/power, and reliability, but also the design of complexity-prone cache coherence protocol and cache reliability solutions. In this paper we make an extensive analysis of existing write policies, namely write-through (WT) and write-back (WB). In the context of the real-time domain, we show that no write policy is superior for all metrics: WT simplifies the design of the coherence and reliability solutions at the cost of performance, WCET, and energy; while WB improves performance and energy results, but complicates cache design. To take the best of each policy, we propose Hybrid Write Policy (HWP) a low-complexity hardware mechanism that reconciles the benefits of WT in terms of simplifying the cache design (e.g. coherence solution) and the benefits of WB in improved average performance and WCET estimates as the pressure on the interconnection network increases. Guaranteed performance results show that HWP scales with core count similar to WB. Likewise, HWP reduces cache energy usage of WT, to levels similar to those of WB. These benefits are obtained while retaining the reduced coherence complexity of WT, in contrast to high coherence costs under WB.

2012 ACM Subject Classification Computer systems organization → Parallel architectures, Computer systems organization → Embedded systems, Computer systems organization → Real-time systems, Computer systems organization → Dependable and fault-tolerant systems and networks

[1] Spanish Ministry of Education, Culture and Sports under the FPU grant FPU15/01394
[2] MINECO and FEDER funds through grant TIN2014-60404-JIN
[3] MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717
[4] European Research Council under EU's H2020 research/innovation programme (grant No. 772773)

30th Euromicro Conference on Real-Time Systems (ECRTS 2018).
Editor: Sebastian Altmeyer; Article No. 3; pp. 3:1–3:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases multilevel caches, real-time systems, multicores, WCET

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.3

Acknowledgements This work was partially funded by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P and the HiPEAC Network of Excellence. It also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773).

1 Introduction

High-performance processors ubiquitously deploy several levels of cache (e.g. IBM POWER 9, Intel Core i7-based systems, and the ARM A Series). This emanates from the positive impact that MLC have on overall system performance. However, MLC design is delicate, not only because it involves high complexity when dealing with coherence, inclusion, and miss policies (among other issues); but also because it can affect key metrics like cycle time, energy/power (and hence temperature), and reliability.

In the real-time domain, the increase in computation needs of critical software across all domains, is driving system designers towards the use of multicores, which necessarily carry the use of MLC systems. For instance, the ARM big.LITTLE (automotive), the Aeroflex Gaisler LEON4 (space), and the NXP T2080 (railway and avionics) architectures comprise two or more levels of cache, the last of which is shared between cores. In addition to average performance, MLC impacts noticeably other metrics specially sensitive to real-time systems: worst-case execution time (WCET) estimates, i.e. guaranteed performance; hardware reliability – particularly critical in the space domain and other harsh environments; and complexity that affects compliance with safety standards (e.g. ISO26262 [17]).

The (cache) write policy determines how writes to lower (L1) cache levels, those closer to the cores, are handled. Under write-through (WT), write operations are performed in the lower cache and are forwarded to the higher (L2) cache level so that both caches hold consistent data. With write back (WB), write operations are only performed in the lower level cache, and the update of the next level is postponed until the cache lines containing the dirty data is evicted from the lower level cache. The write policy impacts the write-miss policy (write-allocate or not write-allocate), the cache coherence solution (e.g. in snooping-based protocols the write miss policy determines – together with the inclusivity protocol – the set of actions to take on a read/write to local and global data), and the reliability solution (e.g. WT usually requires low-overhead parity in lower level caches and ECC in higher level caches, whereas WB requires ECC in dL1 to keep the reliability of data not backed up in L2). Due its remarkable impact on the overall MLC cache design, the write policy affects metrics as important as guaranteed performance, energy/power, and reliability.

Interestingly, each write policy offers a different trade-off among the different metrics and MLC complexity. Hence, the design of the write policy requires finding a balance between them. The latter goes beyond a simple high-performance and real-time classification. Instead, for a given area (e.g. real-time), the particular application domain defines the relevance of each metric and hence, the write policy to use. For instance, in the space domain, due to exposure to radiation, hardware reliability plays a much more important role than in railway. Likewise, performance is much more relevant in automotive, where performance needs are expected to increase by 100x in coming years [4], than in space. In this line, we make the following main contributions:

1. We make an in-depth analysis of both write policies, WT and WB, with emphasis on those metrics of relevance for real-time systems. WT simplifies coherence since most updated data is always in L2, and reliability since the more costly ECC is only needed in L2 with only parity being used in dL1. However, as the pressure on the interconnection (NoC) increases – as a result of integrating more cores – the contention on the NoC generated by writes under WT greatly reduces guaranteed performance (i.e. increases WCET estimates). Further, WT increases energy consumption as each write accesses the NoC and the larger L2. With WB, each write to dL1 does not result in accessing the NoC, with considerable energy consumption reduction; and exceptional WCET reductions. Yet, WB complicates coherence and reliability, increasing cache complexity.
2. We propose Hybrid Write Policy (HWP), a low-overhead mechanism that takes advantage of the good properties of each policy. Building on WT, we attack its average and guaranteed performance issues, with a mechanism that builds on shared/private data classification hardware and applies WT to shared data and WB to private data. HWP removes write-through operations on private data, which in general are the most accessed data, while keeping it for shared data, so cache coherence can be managed as in pure WT caches. At hardware level, in the the Memory Management Unit (MMU) or Memory Protection Unit (MPU), HWP incurs negligible cost for tracking whether memory pages are shared or private and other page properties such as read/write permissions.
3. We evaluate WCET estimation, reliability, energy consumption and coherence cost of HWP. Our results show that for those scenarios in which tasks have limited data sharing, HWP delivers performance similar to WB. Even when the percentage of shared data is as high as 40% HWP remains competitive in all evaluated metrics (other works estimate the percentage of shared data in multiprocessor programs ranges from 25% [13] to 17% [15]). Overall, our design has a simplicity comparable to WT in terms of coherence, while achieving average/guaranteed performance and energy consumption comparable to WB.

The rest of this paper is structured as follows. Section 2 introduces basic concepts of MLC. Section 3 shows some of the main tradeoffs in the design of the cache write policy. Section 4 details our proposal (HWP) in terms of average and guaranteed performance, energy, reliability and coherence control. Section 5 provides empirical evidence of HWP benefits on our evaluation framework. Section 6 presents the most relevant related works. Section 7 summarizes the main conclusions of this work.

2 Background

When designing a multilevel cache hierarchy, see the illustrative example in Figure 4, there are several design choices to be made, which are not independent of each other but quite tightly correlated. In addition to the write policy we have.

With write allocate (WA), on a write miss data is fetched into cache, as it is the case for read misses, and once fetched, the write operation occurs. With no-write allocate (nWA), on a write miss the write operation is simply forwarded to the next cache level (or memory). Both WT and WB can use either of these write-allocation policies, but we only consider WB-WA and WT-nWA caches, since they are the most common choices. Though, our analysis can be extended to other combinations.

The *inclusivity* of the lower cache levels into the upper cache levels (those closer to memory), imposes that all contents in the lower level cache are also included in the upper level cache. Hence, whenever a cache line is evicted from the upper level cache, all cache lines in the lower level cache holding any of the contents of the cache line evicted in the

upper level cache, are also evicted. Under an *exclusivity* approach, cache lines can be stored only in one of the two levels involved. When a new cache line is fetched by the processor, it is typically fetched into the lower level and removed from the upper level. When a cache line is evicted from the lower level it is moved up to the next level. Finally, *non-inclusive* caches are those where no constraint is imposed on whether cache lines are stored in upper or lower cache levels. This is a common choice for instruction caches since they are typically read-only and thus, cache lines can be simply removed on an eviction.

Snooping and directory-based approaches are the most commonly used ones for implementing cache coherence in multicores. For a moderate number of cores, snooping is in general the preferred mechanism because it is faster and much easier to implement and verify [28]. We use it as reference mechanism in this paper. We also assume a bus interconnect, although other interconnect networks would also benefit from our solution. Under snooping, writes can be handled in two different ways: write invalidate or write update. We focus on the write-invalidate MESI (Modified, Exclusive, Shared, Invalid) protocol as one of the most common that also supports write-back caches. We also cover a simple valid/invalid (V/I) protocol, used for WT caches. Under MESI when a snoop write request arrives to cache, the cache invalidates its own copy, if the cache has it. MESI distinguishes between data that is shared (i.e. exists a copy of the same data in another dL1), exclusive (it only exists a copy in the local dL1 and is clean), modified data (i.e. the data only exists in the local dL1 and it is dirty), and invalid data. Coherence is often implemented on top of inclusive caches, so that coherence can be checked in L2 and, only on a read/write request from a core that hits in L2, the dL1 caches of the other cores might be accessed. Under WT with a simple V/I, coherence is completely managed in L2 and, upon a shared cache line write request, it is immediately invalidated in the dL1 caches of the other cores and the data is delivered right away. When dL1 caches are WB and use MESI, on an L2 match a complex process is initiated to invalidate the corresponding dL1 lines, which may be dirty. This stalls the requesting core, with the L2 not accepting further requests until the current one is resolved. This occurs when the potentially dirty line in the dL1 of another core is written back to L2 and invalidated. The fact that accesses may be multi-cycle and non-pipelined to manage coherence imposes the use of complex logic. This may increase design cost and additional power, while significantly affecting critical circuit paths and limit operation frequency. Alternatively, the coherence protocol can be handled at each dL1 cache and the interconnect. With this approach dL1 caches snoop the bus to monitor the activity from the other cores and cores have to expose its activity to the interconnect. This removes the need for using inclusive caches but comes at the expense of an increase in power and complexity in the on-chip interconnect. For instance, in the absence of a shared medium ensuring the in-order delivery of core/memory transactions is difficult [29].

Error correction codes such as single error correction double error detection (SECDED) are inherently complex mechanisms that introduce some delay to encode/decode data. When used in dL1 caches, SECDED can increase cache latency. To prevent so, complex logic is put in place to recover a correct state if data is delivered to upper cache levels unchecked. For L2 caches, SECDED can be more complex since their impact on performance – for instance by making cache access to take an extra cycle – have relatively lower impact than for dL1 caches. When there is no need to correct errors in the dL1 cache, a simple parity mechanism can be used instead of SECDED.

Table 1 Percentage of stores executed by the EEMBC Automotive and MediaBench suites.

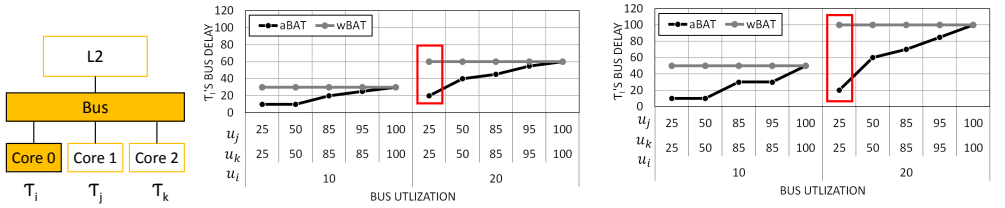
EEMBC	%	EEMBC	%	MediaBench	%	MediaBench	%
a2time	5%	matrix	3%	adpcm.d	13%	mesa.m	12%
aifftr	18%	pntrch	0%	adpcm.e	14%	mesa.o	14%
aifirf	8%	puwmod	12%	epic.d	6%	mesa.t	9%
aiifft	18%	rspeed	14%	epi.e	5%	mpeg2.d	10%
basefp	2%	tblook	6%	g721.d	8%	pegwit.d	6%
bitmnp	11%	ttsprk	4%	g721.e	9%	pegwit.e	6%
cacheb	16%			gsm.d	3%	pgp.d	5%
canrdr	15%			gsm.e	3%	pgp.e	13%
idctrn	8%			jpeg.d	6%	rasta	8%
iirflt	7%			jpeg.e	10%		

3 Tradeoffs in the Design of Cache Write Policy

MLC are one of the main hardware blocks in a multicore architecture devoted to improve performance and reduce the energy/power profile of applications. MLC aim at rapidly and efficiently satisfying data/instruction requests coming from the cores, while maintaining the coherence (i.e. the particular value returned on a read), consistency (i.e. when data is available), reliability (physical integrity) and more recently security (i.e. protection against unwanted/unauthorized actions). The cache write policy, which handles write operations, is at the core of the complexity of MLC since it has a direct impact on the design of other policies. In this section we analyze the impact of WT and WB policies on reliability, inclusivity, and coherence choices. We also analyze their impact on performance (average and guaranteed), reliability, and energy/power. For the latter, the results obtained from several controlled experiments are used as supporting argument.

3.1 Write-Through (WT)

Under WT, each store operation is sent to the L2 so it uses the NoC, which can significantly increase the pressure on it. In the core, the store buffer decouples the commit (finalization) of the stores so that they do not block the pipeline. To that end, once a store reaches the commit/writeback stage, it updates dL1 and in parallel it is placed in the FIFO store buffer allowing the execution to continue. The store is forwarded to L2 when it reaches the head of the store buffer and there is available NoC bandwidth. The store buffer can significantly mitigate the impact of stores in single-core architectures, but rapidly becomes insufficient in multicore. This is better illustrated with an example: let us assume that a bus connects dL1 and L2 caches and each store operation uses it for k cycles. As long as the frequency of stores is (on average) below $1/k$, they will not significantly affect processor performance – unless they are bursted which we do not assume in this simple example. However, in a multicore architecture with N_c cores, as soon as the pressure in the bus increases, the actual duration of a store becomes $k \times N_c$, i.e. $k \times (N_c - 1)$ cycles of contention and k cycles for the bus access. In this scenario, stores become a performance issue as soon as their density reaches $1/(k \times N_c)$. As an illustrative example, Table 1 shows the percentage of store operations executed by EEMBC and Mediabench benchmarks, see Section 5 for more details on the experimental setup. The average percentage of stores is 9%. Further $k \times N_c \in [15, .., 20]$ – and hence it is higher than 1/9, for multicores with 4-8 cores. To make things worse, the percentage of memory operations is growing in emerging data-intensive real-time applications,

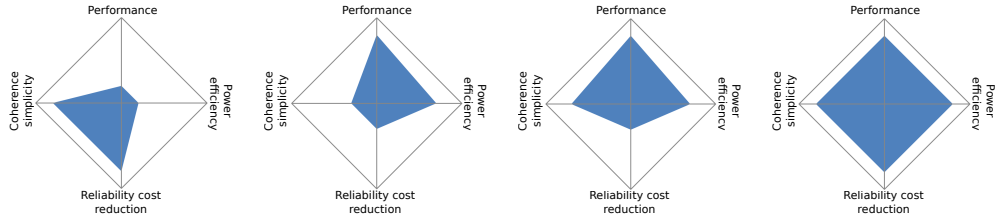


(a) Setup (b) Same duration ($l^i = l^j = l^k$) (c) Different duration ($2 \cdot l^i = l^j = l^k$)

Figure 1 τ_i aBAT and wBAT as a function of its load and its contenders' (τ_j and τ_k) load.

e.g. applications in cars managing data coming from different sensors such as radar, LIDAR, and stereo cameras. Intuitively, this problem can be alleviated by using a crossbar between the dL1 and the L2, at the expense of increased hardware cost. However, this would just shift the problem from the bus to the L2 itself, since L2 access latency is longer than that of the crossbar. Further, to preserve coherence, each store must be allowed to reach any part of the entire L2 cache, which defeats any attempt to mitigate the problem by partitioning the cache space.

The impact of WT on average performance due to NoC contention magnifies for guaranteed performance, causing inflated WCET estimates. This comes from the fact that worst-case time allowances must be done in the WCET estimates to factor in the impact of NoC contention. In general, no assumption can be made on how the requests of the different running tasks are interleaved in the use of the bus. The exception to this are some static timing analysis techniques that keep track of the worst-time when each request from each core can be issued, and hence are able to exactly determine how requests overlap in the access to shared resources [22][14][21]. This, of course, comes at a significant cost, including the increasing effort of making a cycle-accurate model of the MLC system and processor, and increased analysis computation time. Further, this analysis, despite producing (in general) tighter WCET estimates, makes them non time-composable so that any shift in any task requires performing the WCET estimation for all tasks. Hence, to increase time composability and reduce costs, worst-case assumptions are made on how tasks' request are aligned [25, 8, 19]. This is better illustrated with an example. Let us assume a bus connecting the L2 to 3 cores (respectively executing tasks τ_i, τ_j, τ_k) and all bus requests using the bus for the same duration l (shown in Figure 1 (a)). The best overlapping scenario for average Bus Access Time (aBAT) happens when requests of the task under analysis (τ_i) and the contender tasks (τ_j, τ_k, ...) do not overlap as long as the bus utilization of all tasks is below 100%, and when the utilization goes over 100% the minimum overlap happens. For instance if a τ_i uses the bus for 20% of the time and τ_j for 90% of the time, τ_i gets affected only 10% of its time. The worst overlapping scenario for bus access time (wBAT) is assuming that requests from τ_i arrive in the same cycle as the requests from the contender tasks, but τ_i systematically gets the lowest priority. Figure 1(b) shows how worst-case BAT gets much more affected than average BAT due to contention for different scenarios of bus utilization of τ_i and its contenders. We see that wBAT is significantly affected even for low bus utilization. For instance, for utilization $u_i = 20\%$, $u_j = 25\%$, $u_k = 25\%$ for τ_i, τ_j, and τ_k respectively (see red rectangle in Figure 1(b)), τ_i suffers no delay in the best case aBAT and in the worst case it goes to 60% (a 2.4 increase). Further, typically store operations take shorter than load operations accessing the cache (no need to wait for a response), which translates into a scenario in which τ_i requests take shorter than its contenders' request. We see in Figure 1(c), for a scenario in which τ_i requests take half of its contenders, that the impact of contention



(a) WT (b) WB (c) HWP (d) Ideal

Figure 2 Visual comparison of the WT, WB and HWP for the different metrics discussed.

on WCET estimates increase. For instance, for utilization $u_i = 20\%$, $u_j = 25\%$, $u_k = 25\%$ for τ_i, τ_j, and τ_k respectively (see red rectangle in Figure 1(c)), τ_i suffers no delay in aBAT but a 100% in the wBAT.

Continuous store accesses to the L2 cause performance and WCET degradation but can also increase power consumption. Updating values with WT policy implies accessing the bus and L2, even if the core updating the values is the only consumer of this data. This can have a significant impact on the overall power consumption. For example, when running `a2time` from the EEMBC automotive benchmark suite in our reference processor setup (see Figure 4 and Section 5.1), the 14% of the energy consumption comes from the bus and L2.

Under WT, reliability can be handled with reduced overhead. A usual tradeoff consists of using only parity for error detection in dL1 caches, and (usually) apply it at double-word level, that is, using 1 parity bit for 64 data bits (8 bytes). This results in low overhead of around 1.6% (1/64). Furthermore, the operations needed to compute the parity (XOR) can be carried out in parallel and hence are unlikely to affect cycle time. On a parity error, however, hardware support is needed to squash the execution of the instruction that obtained erroneous data and following instructions. On completion, error-free data is fetched from L2 and execution resumes. Alternatively, parity can be checked before delivering the data to remove the need of squash logic. However, this would likely increase cache latency since XOR gates to compute the parity bit may easily need an extra cycle. WT parity-protected dL1 caches are used in combination with SECDED-protected L2 caches. The latter is achieved with ECC that carries an inherent area and logic for its implementation. Typically, SECDED requires 8 code bits per 64 data bits (so $\approx 12.5\%$ extra bits), with negligible impact on L2 performance, since although an additional cycle may be needed to deliver data corrected, this operation is fully-pipelined. Hence, L2 latency may increase by 1 cycle, thus slightly increasing the latency of dL1 read misses, which are generally scarce, but without affecting L2 throughput. Note that on the event of detection of an error in dL1 in a given cache, it is simply discarded and data is fetched from upper cache levels since a correct copy of the data exists in L2 or beyond.

dL1 WT caches simplify coherence management. In particular, dL1 WT caches are made inclusive L2. As a result, when shared data exists in data dL1 (dL1), up-to-date copies of the data is also present in L2. Hence, coherence can be managed in L2 and, upon shared data modifications, the corresponding cores' dL1 caches receive (infrequent) invalidation requests. With WT caches a simple invalidation protocol (V/I)) is enough.

Overall, WT can negatively affect average and worst performance – the latter more intensely– and energy. On the positive side, it can be used with low-overhead coherence and reliability solutions. These properties are summarized in Figure 2(a) in a qualitative manner, with Figure 2(d) showing the ideal scenario.

3.2 Write-Back (WB)

For low core counts, the small average performance improvement of WB over WT does not compensate its additional validation and design costs. However, as the number of cores of multicore real-time systems increases, WB becomes more attractive.

WB significantly reduces the number of bus and L2 accesses compared to WT. Furthermore, since worst-case contention is quite proportional to the number of accesses, WCET estimates are typically much lower with WB than with WT.

WB access count reduction to shared resources decreases the power consumed by those resources. In our setup, the bus and L2 accounts on average for 13% of the system energy, and hence reducing its utilization translates into a non-negligible energy reduction. Also, the need for higher reliability in the data dL1 cache (dL1) increases the power used by the system due to the extra bits and logic needed to implement, for instance, SECDED codes. Finally, since invalidation operations due to shared data accesses may require invalidating dirty lines in dL1, this may cause extra energy consumption to write data back to L2.

When WB is used in the dL1, the data most frequently updated/sensible can be spread between multiple caches (the different dL1 caches and L2). In this scenario, error detection in dL1 and error correction in L2 is not enough, since some data is only updated in dL1 and, upon an error, it could be detected but not corrected. In this case, there are two possible implementations of ECC in dL1, each one with its advantages and drawbacks:

- Under **Data delivery after correction** data is read from dL1, then ECC checked (and eventually data corrected), and finally data is delivered. Unfortunately, checking the ECC code increases access latency by 1 cycle. While such operation can be pipelined, thus not increasing dL1 utilization, the effective latency for data read increases.
- Under **Data delivery before correction** data is read from dL1 and delivered as if it was error-free. In parallel, ECC is checked and, upon an error detection, the affected instruction and subsequent ones need to be squashed. Then, the execution can be resumed using the corrected data. While such process has negligible impact in performance (radiation errors occur only sporadically), the logic for squashing instructions and resuming execution may be complex. However, such logic is analogous to that of WT caches when operating with parity.

With WB caches V/I is not enough because data can be in another state apart from valid or invalid, namely, modified state. Because of this, we will use MESI (an enhancement over MSI) for WB caches. Maintaining cache coherence in multicores with WB dL1 caches requires frequent accesses to other cores' dL1 caches to verify whether shared data is there and, eventually, retrieve them (if dirty) or invalidate them (if the ongoing access is a write or data is not dirty). Depending on the inclusivity of the cache system, we find two possible scenarios (exclusive caches are infrequent so we do not discuss them here):

- **Inclusive**. In an inclusive cache system (the most convenient solution) the updated data is in dL1 or L2, but L2 has all the tags. This means that all coherence requests can go to L2, and only upon a match ask dL1 for the data it needs.
- **Non-inclusive**. If the system is non-inclusive, there is no unique cache that "knows" where all the data is. This means that any request for data has to be communicated to all caches (all the private dL1 and L2), and any cache can answer with the data. This complicates the coherence protocol design. Hence, we disregard this option.

Either case, whenever some data is requested and the L2 experiences a hit on shared data, it must stall the request and block further L2 accesses. Then, the corresponding dL1 caches deliver the data if dirty. Since dirtiness in dL1 caches is not known a priori by the

Table 2 Commercial processors and their characteristics.

Processor	Cores	Freq.	L1 WT?	L1 WB?
ARM Cortex R5	1-2	160MHz	Yes, ECC/parity	Yes, ECC/parity
ARM Cortex M7	1-2	200MHz	Yes, ECC	Yes, ECC
Freescale PowerQUICC	1	250MHz	Yes, ECC	Yes, parity
Freescale P4080	8	1.5GHz	No	Yes, ECC
Cobham LEON 3	2	100MHz	Yes, parity	No
Cobham LEON 4	4	150MHz	Yes, parity	No

L2 cache, it must remain blocked long enough to allow the dirty data to be read from the corresponding dL1 and be sent to L2. Then, the L2 can update its contents, deliver the data and hence, serve the request. However, the complexity of the logic to manage all this process synchronously and across multiple cycles and components may affect critical circuit paths, which can carry a reduction of the operating frequency.

Figure 2(b) presents in a graphical manner the assessment we have done on WB. We can see that while WT is better in reliability and coherence simplicity, it performs worse on performance (both average and worst-case) and power.

3.3 Cache Write Policy in Some Commercial Architectures

To better illustrate the quandary chip vendors face when selecting the write policy, we have analyzed the miss policy of several commercial processors[5].

The ARM Cortex R5 [3] is a 1 (or 2) core processor that implements both WB and WT in the dL1 cache, both with parity and ECC. This means that either policy can be selected in a configuration register. The ARM Cortex M7 [1] is a low-performance processor. Like the previous one, it implements both write policies in the dL1 cache, but it only has ECC in the L2 cache. ARM acknowledges that using dL1 ECC may have an impact on operating frequency due to the XOR trees for the ECC when getting the data from the cache. Thus, depending on the particular chip implementation of the ARM IP processor we might have to decrease maximum operating frequency or require two cycles to access the dL1 to support ECC in the dL1 and have the possibility of recovering from errors in the cache when WB is enabled. Hence, despite in general WB caches perform better the strong reliability constraints in safety-critical systems and the associated overheads incurred due to implementing ECC in WB caches makes chip vendors offer the users the possibility to choose between WT/WB according to the needs of their application. However, this forces chip vendors to carry with the effort and responsibility to implement and validate both.

The Freescale PowerQUICC [32] implements WB in the dL1 with parity and the L2 with ECC. This lead to a system where not all cache bit-flips can be recovered. In that respect, Freescale states that the probability of errors is so low that the target application domain should accept the possibility of having “unrecoverable" errors.

The Cobham Gaisler LEON3 [12] is a dual-core running at 100MHz, with a 5 stage in-order pipeline. It is designed for critical real-time systems, and implements WT in the dL1 cache, so that reliability can be handled in L2 with more robust ECC. The LEON4 [10] comprises with 4 cores running at 150MHz with a 7 stage pipeline. It has the same critical real-time systems scope as its predecessor, and the same write policies in the dL1.

[5] Core and frequency numbers have been obtained from specific processor implementations [36, 35, 26].

WT has been widely implemented in the last level of private caches (mainly in dL1) due to its simplicity (no need for reliability and simple coherence) and its acceptable single-core performance. However, in future multi- and many-cores, the increased number of accesses to shared resources will cause a dramatic increase in average execution time and the WCET estimates. WB caches have performance and energy consumption benefits over WT in mid to high core count processors. However, this performance comes at a complexity cost in the coherence protocol mainly and, to a lower extent, in the reliability mechanisms.

4 Hybrid Write Policy (HWP)

HWP low-overhead approach addresses WT average and guaranteed performance issues while reducing overheads w.r.t. WB. HWP eliminates the additional cost of coherence for WB caches and, simultaneously, keeps WT operations limited to a small fraction of write operations so that efficiency is close to that of WB caches, see Figure 2(c).

In order to reach its goals HWP builds on the following observations. First, cache coherence management with WB caches is costly and complex because cache lines accessed may reside dirty in local dL1 caches. Second, private data is not affected by cache coherence, so conceptually it is irrelevant whether such data is dirty or not in dL1 caches. And third, a significant percentage of memory data is only accessed by one processor (also in parallel applications) and, thus, does not require keeping coherence (e.g. 75% of the access are reported as private in [13] and around 83% in [15]).

From those observations, we design a new policy (HWP) that manages private data as in WB caches and shared data as in WT caches. With HWP, memory contents are classified at page granularity as either shared or private, which has been shown to be a very convenient granularity for private/shared data classification [15, 6]. In particular, as long as a page contains any shared data, it is (pessimistically) classified as shared. Otherwise, it is classified as private. On a write to shared data, HWP writes it through to L2 cache (a la WT). Meanwhile write operations to private pages are not propagated to L2 (a la WB), hence decreasing contention in the access to L2.

Next, we discuss the key characteristics and implementation details of HWP, with emphasis on how to classify data as private or shared (and the appropriate granularity to do so), how to check whether data is shared or private to decide whether to proceed as in a WT or WB cache, how cache coherence needs to be managed, what the reliability implications are, and how contention in the access to L2 is mitigated.

4.1 Data Classification

Orthogonally to HWP, a mechanism is needed to classify data as private or shared. Techniques exist to that end, with some of them [15] already integrated on a real hardware platform (LEON3 processor) and Linux, providing evidence of its feasibility. Interestingly, private/share data classification can be performed at different levels (e.g. cache line size).

Private/shared information can be managed at fine granularity (e.g. cache line level). This would allow a much finer classification but at the cost of higher area and energy overheads [15]. Additionally, performing the shared/private classification at page granularity makes it possible using OS functionality to reduce hardware implementation overheads [6].

Ho et al. [15] and Cuesta et al. [6] show that the most convenient granularity to classify data is page level. With this solution, whenever a piece of data is shared between two cores, the whole page in which the data is is marked as shared. Hence, this solution pessimistically assumes that all data in a shared page is shared. As part of that solution, the information on

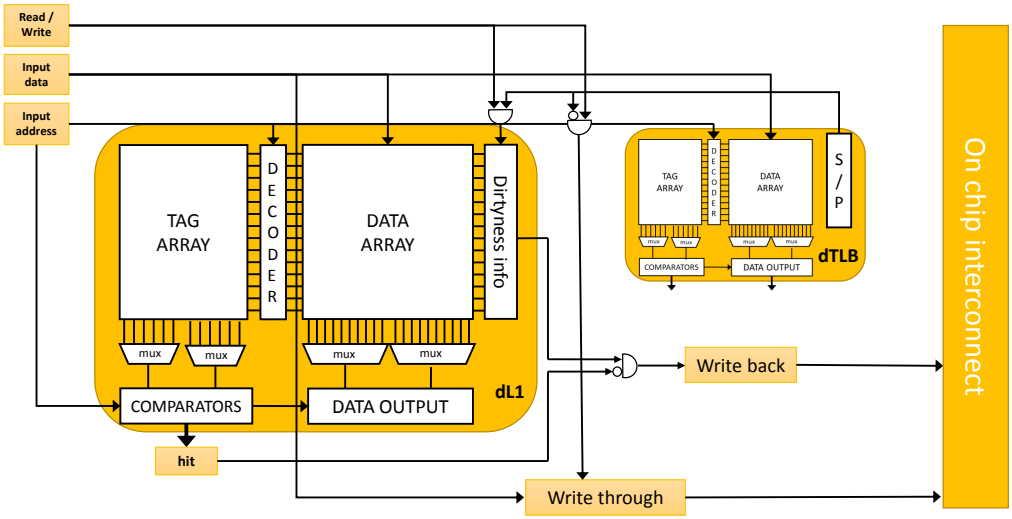


Figure 3 Schematic of HWP cache access protocol.

private/share information can be stored in the Memory Protection Unit (MPU) or Memory Management Unit (MMU) for each page along with other information such as whether pages are user-level or supervisor-level, whether they are read/write or read-only, and whether they are cacheable or not. Such information is often cached in the Translation Lookaside Buffer (dTLB) together with address translation. In most processors dTLBs are accessed in parallel with dL1 caches for fast address translation and for verification of the permissions to read/write in specific memory pages. Hence, they can store private/shared information.

In real-time systems an alternative approach to those hardware approaches is possible with software address space partitioning. Many OS use address spaces (i.e. a range of addresses) to map specific I/O devices. Also RTOS like PikeOS use separate address spaces to implement resource partition. Furthermore, in the automotive domain, AURIX architectures come equipped with caches and different memory types (e.g. flash, ram). From the software side, address ranges are defined to map data/instructions to the desired memory and or to make data cacheable or non-cacheable. Hence, address space can be partitioned assigning a particular address range to shared data.

The main disadvantage of dynamic hardware solutions is that data re-classification is needed. This happens, for instance, when a page is first loaded by one core (hence classified as private) and then accessed by another core (being reclassified as shared). This does not only create predictability issues in real-time systems, but it also adds complexity to HWP, including writing through all data (dL1 lines) of this page in the owner core, while managed those same data with WB in the other cores. This complexity is avoided with the classification based on software address partitioning, *which is the solution we assume in this paper*, without loss of generality.

4.2 Private/Shared Data Management

The way in which data is accessed under HWP varies depending on whether data is shared or private. This is graphically illustrated in Figure 3.

On a load/store access, the dL1 and the dTLB are accessed in parallel. In case of a dTLB miss, it is served first before proceeding with the access, as done regularly in most processors. Note that address translation is typically needed before accessing the L2. Therefore, while

Table 3 Timing of a dL1 hit (dTLB hit) under HWP.

		cycle 1	cycle 2
LOAD		Read dL1, Read dTLB	
STORE	Private	Write dL1, Read dTLB	Update dirtiness bit
	Shared	Write dL1, Read dTLB	Write L2

serialization of dTLB misses and dL1 accesses may be unnecessary for some dL1 hits, dTLB miss rates are usually extremely low, and their occurrence together with dL1 hits is even more unlikely since this can only occur if data from the page has been fetched and sufficient evictions occurred in the dTLB but not in the dL1.

4.2.1 Hit in dL1

In case of a dL1 hit (and dTLB hit), the shared/private information determines whether the line hit needs to be marked as dirty or not. If the line belongs to a private page ($S/P = 0$) and the access is a write operation ($W/R = 1$), the dirtiness bit is set. The separation of data and dirtiness information poses no issue since dirtiness information can be accessed systematically one cycle after, as it is only needed in case of a miss to decide whether the evicted cache line needs being written back. Also, in case of dL1/dTLB hit, if the line belongs to a shared page ($S/P = 1$) and the access is a write operation, data is written through L2 as in a regular WT MLC.

In terms of timing, Table 3 shows the different possible scenarios and their timing. After the processor request, regardless of whether it is a read or a write, both the dL1 and the dTLB are accessed in parallel. In the case of a read, at the end of the first cycle the data is available and is served to the processor. In the case of a write, at the end of the first cycle the dTLB determines whether it is a write to a shared or private page. If the store targets a private page, the dirtiness bit is updated in the dL1 cache in the second cycle, and the request is completed. However, if it is a write to a shared page, a write request is issued to the L2.

4.2.2 Miss in dL1

On a dL1 miss, WT management is performed as for hits. However, if the miss corresponds to a read operation ($W/R = 0$) or the address is private ($S/P = 0$), the line is fetched from L2 and allocated in the dL1 data cache. Note that we assume the usual case where WT implements no-write-allocate (nWA) policy on write misses, whereas WB implements write-allocate (WA). Different write allocate policies could be implemented such as, for instance, WA (or nWA) regardless of the privateness of the data accessed.

In Table 4 we see the different scenarios that can happen with a dL1 miss. In the first cycle, both the dL1 and the dTLB are accessed in parallel. At the end of the cycle, if the line to be evicted is dirty, the dL1 sends a dirty eviction request to the upper level. In the load scenario, the next cycle (3 if the line was dirty, 2 if it was clean) the line is requested to the L2. After n cycles, the cache line arrives to the dL1 and the data is be available. In the case of a store private, it also requests the line to the L2. When the answer comes, it updates the dL1 and update the dirty bit (allocate on private data). Finally, on a store to a shared line, a request for the write is sent to the L2, and no update occurs in dL1 (no allocate for shared data).

Table 4 Timing of a dL1 miss (dTLB hit) under HWP.

		cycle 1	cycle 2	cycle 3	...	cycle n+3
LOAD		Read dL1, Read dTLB	if dirty -> Eviction	Request line L2		Read dL1
STORE	Private	Write dL1, Read dTLB	if dirty -> Eviction	Request line L2		Write dL1, Update dirtiness bit
	Shared	Write dL1, Read dTLB	if dirty -> Eviction	Write L2		

4.3 Non-Functional Metrics

This section makes a qualitative assessment of the benefits of HWP over WT and WB. Quantitative comparisons are carried out in the Section 5.

Under HWP, shared data is consistently stored in L2, making that all shared data in dL1 caches is necessarily non-dirty. As a result, with HWP coherence is managed as in the case of pure WT caches, hence keeping its low- cost and complexity benefits and avoiding the overheads related to WB caches. With HWP V/I is enough, as for WT, because the shared data will always be updated in a single place (L2), so we do not need a Modified state in the dL1 to keep track of who has the most updated data.

Since shared contents are written through to L2, the fraction of dirty dL1 cache contents is smaller than in pure WB caches. Yet some dL1 cache contents can be dirty. Hence, error correction capabilities are still required in dL1, as in the case of pure WB caches. A simple software solution to reduce the associated costs consists in marking the pages storing error-sensitive data as shared. This way the only data that could be lost would be the private one. However, for critical applications, the same reliability technique used in WT (SECDED in dL1) can be used.

Under WT, performance issues relate to contention in the NoC and the L2 due to write-through stores. With HWP, this problem is alleviated, restricting write throughs to stores to shared data. Obviously, the lower the number of accesses to shared data, the lower the number of WT operations, and hence, the lower the contention and the lower the sensitivity to contention. In general, programs are designed to reduce access count to shared data (25% [13] and 17% [15]), which usually carries a serialization of tasks.

In general, power consumption relates to the activity performed (dynamic power) and execution time (static power). By limiting the number of WT operations, dynamic power is reduced drastically w.r.t. pure WT designs. By reducing contention, execution time is also lower than for pure WT designs, thus reducing static power.

Overall, our HWP hybrid cache design offers a globally better tradeoff than WB and WT. This is illustrated in Figure 2(c). As shown, our design offers performance and power close to that of WB, with similar reliability overheads, but much lower complexity for the management of shared data. When compared against WT, coherence management cost is identical, performance and power are much better, and only reliability costs are higher.

5 Evaluation

In this section we quantitatively assess the benefits of HWP over conventional write policies (WT and WB). We use the metrics presented in previous sections, namely, guaranteed and average performance, energy consumption, coherence overhead, and reliability.

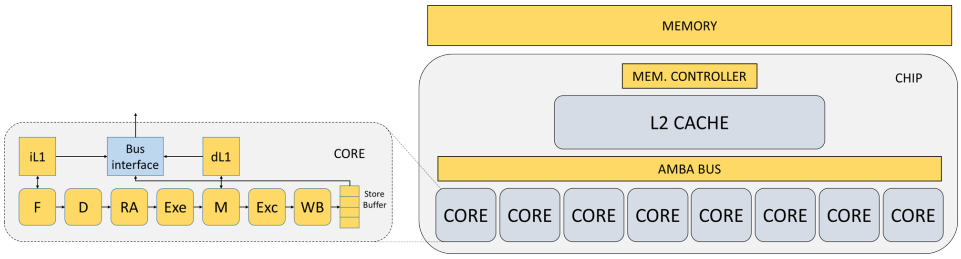


Figure 4 Block diagram of the main elements of our NGMP-based 8-core architecture.

Table 5 Benchmarks in EEMBC Automotive and MediaBench we use in this paper.

suite	List of benchmarks
EEMBC Auto	`a2time`, `aifftr`, `aifirf`, `aiifft`, `basefp`, `bitmnp`, `cacheb`, `canrdr` `idctrn`, `iirflt`, `matrix`, `pntrch`, `puwmod`, `rspeed`, `tblook`, `ttsprk`
Media-Bench	`adpcm.d`, `adpcm.e`, `epic.d`, `epic.e`, `g721.d`, `g721.e`, `gsm.d`, `gsm.e`, `jpeg.d`, `jpeg.e`, `mesa.m`, `mesa.o`, `mesa.t`, `mpeg2.d`, `pegwit.d`, `pegwit.e`, `pgp.d`, `pgp.e`, `rasta`

5.1 Reference Architecture and Benchmarks

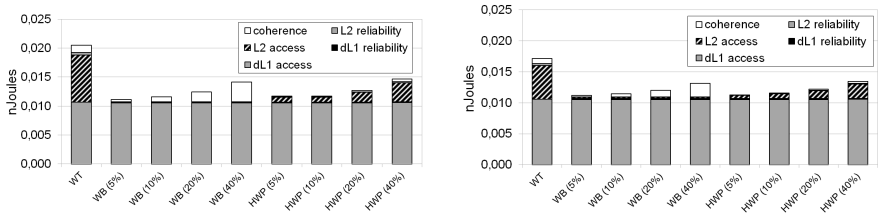
We use a simulation environment based on the cycle-accurate SoCLib [33] framework to model the architecture of the Cobham Gaisler's Next Generation Multipurose Processor (NGMP), as a representative of current bus-connected multicore MPSoC. The main difference lies in that we scale the number of cores from 1 to 8 in our experiments (See Figure 4) to assess the impact on the different metrics, while the NGMP specifically features 4 cores. Each processor implements a SPARC V8 architecture [11]. Each LEON4 core comprises seven stages: fetch (F), decode (D), register access (RA), execution of non-memory operations (Exe), dL1 access (M), Exceptions (Exc) and write back (WB). The execution units are equipped with an integer and a floating-point unit (FPU). Each core has its own private instruction (il1) and data (dL1) caches that are 16KB, 4-way with 32-byte lines. Processors are connected by a shared on-chip round-robin arbitrated AHB processor bus to a shared L2 cache and memory. The shared second level (L2) cache is split among cores, each receiving one way of the L2. All caches use LRU replacement policy.

We evaluate a large subset of the EEMBC automotive [27] suite comprising common critical real-time applications in automotive systems and MediaBench [20] comprising embedded applications such as multimedia and communications. The benchmarks we use from both suites are listed in Table 5. We create several scenarios in which we vary the percentage of accesses targeting the address range for shared data.

5.2 Energy

As presented in Section 3 each cache write policy carries side effects on the write-miss policy, the reliability solution, inclusivity, and the coherence solution. This affects the set of activities carried out by each task, the energy cost of each activity and hence the overall energy profile of each task. Further, the complexity of each write policy varies which affects its 'intrinsic' energy consumption.

We assess the energy usage under each policy using CACTI [24], the state-of-the-art integrated model for cache and memory access time, cycle time, area, leakage and dynamic power consumption, configured with the NGMP cache parameters. With CACTI we break-



(a) `EEMBC` (b) `Mediabench`

Figure 5 Average energy breakdown per cache access for EEMBC and Mediabench.

down the energy usage of each cache access into 5 components: dL1 access, dL1 reliability, L2 access, L2 reliability, and coherence.

We present the average cache access energy consumption, across all EEMBC and Mediabench benchmark suites, in Figures 5 (a) and 5 (b) respectively. The difference across individual programs in each benchmark are not relevant, and hence are not shown. We compare WT, WB and HWP; and for the latter two, we assume three different scenarios depending on the percentage of accesses to shared data: 5%, 10%, 20% and 40%. Note that WT results do not depend on the percentage of shared data, since all writes go to L2.

We observe that the dL1 energy usage for an access is roughly the same for all write policies. The difference in the energy of the dL1 reliability solution is small, with WT having the lowest value due to the use of simple parity instead of ECC (used by WB and HWP).

We also observe that the lowest access energy profile is obtained for WB and HWT. In the case of WB, there are few L2 accesses, since stores do not access L2 every time, while the load access rate to the L2 is relatively low. HWP has a higher L2 access rate than WB since it writes shared data directly to the L2.

On the coherence side, WB has an increased amount of coherence-related messages as the shared data increases. Taking into account all components, WB and HWP consume roughly the same energy per access for a given ratio of accesses to shared data. Both show approximately a 42-50% per access energy reduction (depending on the percentage of shared data) with respect to WT. To sum up, when comparing the different write policies on the energy aspect, HWP has the same reduced energy consumption as WB compared to WT (up to 50%), but without the coherence complexity inherent to WB, as presented in Section 4.

5.3 Guaranteed Performance

WCET estimation is one of the most critical metrics for real-time systems, since it determines the guaranteed performance that the system can deliver. As presented before, WCET estimation is challenged by the use of multicores due to contention delay suffered by tasks.

In order to assess the benefits on WCET estimate reduction of HWP, we have created 1-, 2-, 4- and 8-task workloads, as presented in Table 6. Workloads have been generated using benchmarks from the EEMBC automotive suite (eembc1.X, eembc2.X) and from the MediaBench suite (media1.X, media2.X). Across workloads, the first task in each workload, the one for which WCET estimates are produced, comprise at least one benchmark with at most a 5% of stores, and at least one benchmark with at least a 13% of stores. The rest of the new benchmarks in the workload are selected randomly.

Modeling multicore contention is a concern for timing validation and verification as witnessed by a notable amount of works on the topic, summarized in [9]. Many measurement-based approaches – the most extended industrial practice – build on the availability of

Table 6 Benchmark mixes used to assess WCET estimates under different core counts.

Mix	main	cont1	cont2	cont3	cont4	cont5	cont6	cont7
eembc1.1	bitmnp							
eembc1.2	puwmod							
media1.1	g721.d							
media1.2	jpeg.d							
eembc2.1	bitmnp	a2time						
eembc2.2	puwmod	aifftr						
media2.1	g721.d	adpcm.d						
media2.2	jpeg.d	adpcm.e						
eembc4.1	bitmnp	a2time	matrix	rspeed				
eembc4.2	puwmod	aifftr	idctrn	ttsprk				
media4.1	g721.d	adpcm.d	gsm.d	pegwit.d				
media4.2	jpeg.d	adpcm.e	g721.e	pgp.d				
eembc8.1	bitmnp	a2time	matrix	rspeed	tblook	canrdr	aifirf	aifftr
eembc8.2	puwmod	aifftr	idctrn	ttsprk	basefp	cacheb	tblook	ttsprk
media8.1	g721.d	adpcm.d	gsm.d	pegwit.d	g721.d	pegwit.d	gsm.e	pgp.d
media8.2	jpeg.d	adpcm.e	g721.e	pgp.d	adpcm.e	jpeg.d	gsm.e	adpcm.e

performance monitoring counters (PMCs) [23, 25, 7, 18, 8]. From those we build on [18] since it captures the number of requests each core performs to the shared resources. This results in *partially time composable* WCET estimates, rather than *fully-time composable* ones that result from assuming that every single request of the task under analysis is delayed regardless of the load contenders put on the shared resources.

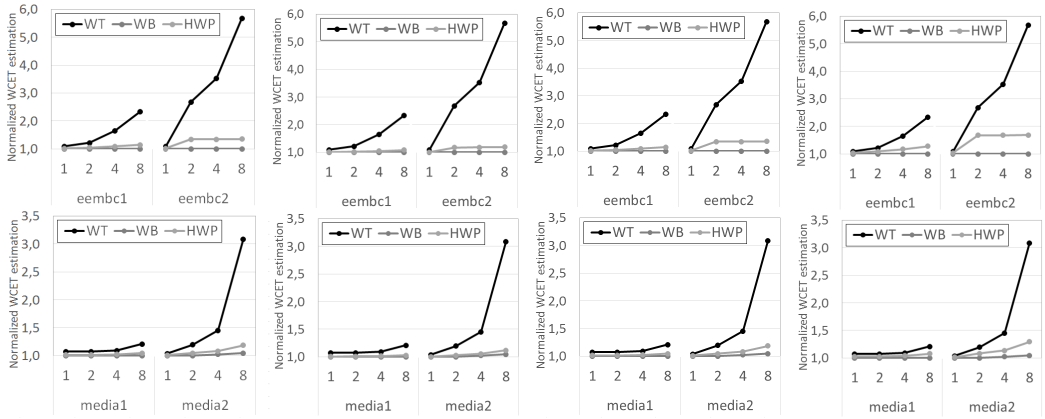
We illustrate the model [18] with a small example comprising one task under analysis or τ_a and a contender task or τ_b. When τ_b has more requests than τ_a, each request of τ_b is assumed to delay the requests of τ_a. The worst-case contention that τ_b can cause on τ_a, i.e. $\Delta^{cont}_{b\rightarrow a}$, is computed according to Equation 1, where n^t_b is the number of τ_b requests of type t and lat^t is the latency of that request type. Note that the model makes the worst case assumption of no overlap of requests, so each τ_b's request delays τ_a by its latency, i.e. lat^t.

$$\Delta^{cont}_{b\rightarrow a} = \sum_{t\in\mathcal{T}} min(n_a, n^t_b) \times lat^t \tag{1}$$

In our case the request types are $\mathcal{T} = \{L2h, L2m, s2h, s2m\}$ corresponding to loads hitting and missing in the L2 cache, and stores hitting and missing in the L2 cache respectively, which can be tracked with existing PMCs [11]. The corresponding latency of each of these event type is [18] (in processor cyles): $lat^{L2h} = 9$, $lat^{L2m} = 7$, $lat^{s2h} = 1$, and $lat^{s2m} = 1$.

Note that it does not matter the type of τ_a requests but just it overall number $n_a = \sum_{t\in\mathcal{T}} n^t_a$. That is, the contention τ_a suffers depends on its total number of requests and the number of requests of each type of its contenders (τ_b in this case). The model factors in the case when τ_b has fewer accesses than τ_a that results in some τ_a requests not being delayed by any request from τ_b. The approach presented in Equation 1 for τ_b is followed for all the $Nc-1$ tasks simultaneously running with τ_a, where Nc is the number of cores. The reader is referred to [18] for more details.

Figure 6 shows the WCET estimate obtained for the first task under each cache write policy. WCET estimates are shown as the number of cores varies from 1 to 8. In order to simplify the comparison, all WCET estimates are normalized to the WCET estimate of



(a) 0% shared (b) 10% shared (c) 20% shared (d) 40% shared

Figure 6 Normalized WCET estimate for the first task in the workload under different core counts and percentage of shared data for the different write policies.

the first task when run in isolation under WB. We see that in all cases the tightest WCET estimates are obtained with WB. HWP obtains comparable results to those of WB and much better than those for WT. The latter gets rapidly worse as the core count increases. Note that Figures 6 (a), (b), (c), and (d) are not directly comparable, since for each figure WCET estimates are normalized to that of WB when the task runs on isolation.

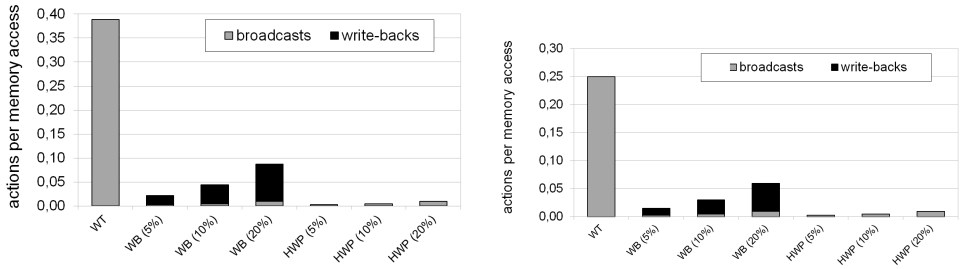
We also see that WT is not affected by the percentage of shared data, since it always updates the L2 regardless if the data is shared or not. WB does not show meaningful variations either, while HWP has small variations (mainly for eembc2 and media2). In all cases HWP is significantly better than WT.

Across all shared-data scenarios for WT we can observe that:

- Mix eembc2 suffers a significant increase in WCET estimates (more than 5x in the 8 core configuration). This is due to the combination of memory instructions the program under analysis executes (30% of all instructions) and the number of stores the competing tasks have (9% of all instructions on average).
- Mixes eembc1 and media2, have lower, yet significant, WCET increases (more than 2x and 3x respectively). This is caused by the combination of the two metrics just mentioned is lower than that of eembc2.
- Finally, media1 has a small WCET estimate increase due to a lower number of memory instructions executed by the main program (23%) and a lower percentage of stores in the challenger tasks (6% on average).

WB is the write policy with lower WCET estimate performance penalty. WB causes a small increase in WCET estimates even when we have 40% of data shared (higher than what is usually found in parallel applications [13, 15]). This is so, because only data requested by other cores is exchange via the bus.

HWP lies in between WT and WB, though it is much closer to WB. For eembc1 and media1, the WCET estimate is remarkably low. This is also contributed by low percentage of memory instructions combined with the low percentage of stores in the challenger tasks. For eembc2 and media2, HWP suffers high increase in WCET estimates in the 40% shared data scenario: despite HWP reduces the pressure on the bus, (i) the high percentage of share data, (ii) the high percentage of memory instructions these benchmark mixes execute (30% and 23% respectively), and (iii) the number of instructions that are stores in the competing



(a) `EEMBC` (b) `Mediabench`

Figure 7 Number of broadcasts and write-backs per memory access.

tasks (8 and 9% respectively), cause the pressure on the bus to increase. Yet, HWP stills performs better than WT, specially in high-core setups (4-8), where WT grows to 3-6x and HWP only grows to less than 2x in the worst setups. The penalty difference with WB as the number of cores and shared data increase is mainly due to the fact that all accesses to shared data is sent to the L2 (write-through on shared data), without the need of another core requesting the data. This means that the same core could write several times directly to L2 without another core requesting the data in between.

To sum up, HWP obtains similar WCET estimates to WB, but significantly smaller than WT (up to 5x) in multi-core setups. This difference in WCET estimates increases significantly with the number of cores being used.

5.4 Coherence

The write policy impacts the selection of the coherence solution. With WT caches a simple invalidation protocol V/I is enough, while for WB caches a more complex policy such as MESI is required. For HWP, an invalidation protocol such as the one used in WT is enough.

The potential impact of the coherence protocol, in particular MESI, is two-fold. First, the complexity of its design, implementation, and validation. And second, its impact on performance since the number of messages to exchange between processors and the L2 cache to maintain coherence.

Since the complexity has been qualitatively assessed in Sections 3 and 4, here we focus on the number of messages that will be sent in every coherence protocol as a proxy to coherence performance overheads. In particular we focus on the invalidation messages and the number of write-backs caused because of coherence (not due to cache capacity issues).

Figure 7 shows the average number of coherence messages per memory access for EEMBC (a) and Mediabench (b). We evaluate the 3 write policies: WT, WB and HWP; considering 5%, 10% and 20% of shared accesses in the last two policies. The number of invalidations in WT is high in both benchmark suites because every write access requires that an invalidation message is sent to the bus, since any other private cache can have a copy of the data. For WB and HWP the number of invalidations is much smaller, since the cache directory tracks the core having a copy of each cache line, and only shared data that actually is in private caches will be invalidated.

The other coherence metric we analyze is the number of write-backs related to coherence, which only happen for the WB policy. This occurs when a core c_0 modifies some data in its private dL1 and another core c_1 wants to access that data. Since the L2 knows that c_0 has this data in a Modified state, the L2 asks c_0 to write back the modified data to L2, and then

the L2 sends it to c_1. In the WT policy, the L2 always has the most updated values, so there are no write-backs due to coherence. Likewise, HWP only writes back private data, treating shared data like WT, so there are no write-backs due to coherence either.

Note that while the trend for the coherence cost shown in this section is similar to that of energy (Figure 5), the absolute values are not the same. This is so because the energy cost of a write-back is higher than that of an invalidation. As a result, when comparing WT to WB/HWP, the energy consumed in coherence is not that high as the number of messages as shown in this section.

Overall, HWP offers the best of WT and WB in terms of coherence: it generates as little invalidations as WB without the coherence related write-backs of WB.

5.5 Reliability

We assume that caches are able to detect and correct single-bit upsets (SBU), while multi-bit upsets (MBU) may occur when their probability is high enough. We assume that solutions such as word interleaving[6] are applied so that a N-bit MBU becomes N SBUs. Hence, the criteria to assess reliability consists of whether designs are able to detect and correct single-bit errors. Note that such reliability criteria are already implemented in processors targeting the highest criticality levels in the space [11] and automotive [2] domains.

Since in WT all the updated data is always in L2, only parity is required in dL1 to detect single-bit errors given that correct data can be retrieved from L2. WB and HWP allow dirty data in the dL1 cache, and thus they require error correction capabilities in dL1, such as SEC-DED. The L2 cache always implements SEC-DED, since there can be dirty data at this cache level when using all policies.

The difference in the reliability technique used in dL1 has limited impact on area. Parity, used in WT, imposes a 1.6% increase in the number of cells needed (1 bit per 64-bit word), as well as few XOR gates and a comparator. SEC-DEC, used in WB and HWP, increases by 12.5% the number of cells (8 bits per 64-bit word), and also adds extra XOR gates and comparators [16]. Note that the relative area of dL1 cache w.r.t. L2 cache is typically low, and all write policies implement SEC-DED in L2, thus lowering the relative additional cost of SEC-DED vs parity in dL1 when put in the context of the complete cache system.

This section complements the comparison that has been made in Sections 3 and 4.3.

6 Related Work

Relevant related works relate to WB caches and their use in real-time systems, private/shared data classification mechanisms, models for computing WCET estimates for multi-core contention, and the use of other hybrid techniques for high-performance computing.

Due to the recent interest in the use of WB caches for critical real-time systems, mainly due to its potential increase in guaranteed performance, some works [34, 5] have studied static WCET analyses of this write policy, since it is more challenging than for WT caches. Authors in [34] propose an eviction-focused technique, analyzing for each cache miss if it could result in a write-back in order to estimate the WCET. In a more recent work [5], a new method has been proposed to complement the previous work by using a store-focused technique. This method consists in checking whether a store may transform a currently

[6] Interleaving K words at bit level ensures that bits of a given word, and hence protected with the same parity/ECC code, are at a distance of at least K bits.

clean line into dirty, and hence result in a write-back later on. Those techniques can be retargeted to capture HWP to tighten WCET estimates over WB/WT. In this line, previous works [31] also propose new cache systems that take into account shared/private data to improve WCET estimates, but with more radical changes required in the architecture.

Regarding private/shared data classification, different methods and hardware designs based on them have been proposed [13, 15]. Some authors [13] classify the different types of cache access patterns, and use such classification to implement a specific distributed cache design. Authors study the percentage of data that is private, shared/read-only and shared/modified. In [15] the authors propose a dynamic classification of shared and private pages. This technique needs some WB mechanism when a page changes its status from private to shared. While this technique may also improve performance over WT, it also has to deal with the coherence complexity of WB.

WCET estimate computation in multicores has been subject to intense study [23, 25, 7, 18, 8]. In [18, 8], the authors propose techniques for computing partially time composable Execution Time Bounds for bus accesses based on the number of requests the contenders can generate, regardless of when they access the bus. These technique provides tighter WCET estimates than simpler fully time composable models that always assume the worst case on a bus access. We have built on these techniques for WCET estimation.

Techniques for a hybrid approach on coherence management have been studied in high-performance domains [30, 6]. In [30], the authors implement a similar technique to [15] that dynamically changes the status of memory pages from private (default) to shared when they are accessed by more than one core. In [6], the authors propose a similar technique to differentiate private and shared pages at OS level, thus reducing the size of cache directories since they do not need to keep track of private lines. However, in these works there is still a non-trivial coherence mechanism with transient states, while our proposal targets a simpler (static) coherence mechanism.

7 Conclusions

The relentless trend towards the adoption of multilevel caches in real-time systems is a fact, in the line of high-performance systems. Our analysis of the write miss policy shows that WT simplifies coherence and reliability, while WB performs better in performance and energy. From the analysis we propose a new Hybrid Write Policy (HWP) that discriminates among shared and private data to smartly write through dL1 data or keep it dirty in dL1. Experimental results show that HWP results in remarkably better guaranteed performance than WT. HWP results for energy consumption per memory access improve those of WT. In terms of complexity of the coherence protocol, HWP implements a simple Valid/Invalid protocol like WT, compared to the complex MESI protocol used in WB.

References

1 ARM. ARM Cortex-M7 processor. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B_cortex_m7_trm.pdf.

2 ARM. Arm cortex-r series processors specification. http://infocenter.arm.com/help/topic/com.arm.doc.set.cortexr/index.html.

3 ARM. ARM Cortex R5 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0460d/DDI0460D_cortex_r5_r1p2_trm.pdf.

4 ARM. ARM expects vehicle compute performance to increase 100x in next decade. https://www.arm.com/about/newsroom/

arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php, 2015.

5 T. Blaß, S. Hahn, and J. Reineke. Write-back caches in WCET analysis. In *ECRTS*, 2017.

6 B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *ISCA*, 2011.

7 D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, 2011.

8 E. Díaz, M. Fernández, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and F. J. Cazorla. MC2: Multicore and cache analaysis via deterministic and probability jitter bounding. In *ADA-Europe*, 2017.

9 G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *WCET Workshop*, 2014.

10 Cobham Gaisler. LEON4-N2X data sheet and user's manual. http://www.gaisler.com/doc/LEON4-N2X-DS.pdf.

11 Cobham Gaisler. NGMP preliminary datasheet version 2.1. http://microelectronics.esa.int/gr740/LEON4-NGMP-DRAFT-2-1.pdf.

12 Cobham Gaisler. UT699 32-bit fault-tolerant SPARC V8/LEON 3FT processor data sheet. http://www.gaisler.com/doc/gr712rc-datasheet.pdf.

13 N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 2009.

14 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009.

15 N. Ho, I. I. Ashraf, P. Kaufmann, and M. Platzner. Accurate private/shared classification of memory accesses: a run-time analysis system for the LEON3 multi-core processor. In *DATE*, 2017.

16 M. Y. Hsiao. A class of optimal minimum odd-weight-column SEC-DED Codes. In *IBM Journal of Research and Development*, 1970.

17 International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

18 J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Bounding resource contention interference in the next-generation microprocessor (NGMP). In *ERTS*, 2015.

19 H. Kim, Dionisio de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS*, 2014.

20 Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, 1997.

21 B. Lesage, D. Hardy, and I. Puaut. Shared data caches conflicts reduction for WCET computation in multi-core architectures. In *RTNS*, 2010.

22 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

23 T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald. Methods for modeling resource contention on simultaneous multithreading processors. In *IEEE ICCD*, 2005.

24 N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. In *HP Tech Report HPL-2009-85*, 2009.

25 J. Nowotsch, M. Paulitsch, D.B. Uhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

26 NXP. MPC8245 integrated processor hardware specifications. https://www.nxp.com/docs/en/data-sheet/MPC8245EC.pdf.

27 J. Poovey. *Characterization of the EEMBC Benchmark Suite*, 2007.

28 A. Roca, C. Hernandez, M. Lodde, and J. Flich. Area-efficient snoopy-aware NoC design for high-performance chip multiprocessor systems. In *Computers & Electrical Engineering*, 2015.

29 S. Rodrigo, J. Flich, J. Duato, and M. Hummel. Efficient unicast and multicast support for CMPs. In *MICRO*, 2008.

30 A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *PACT*, 2012.

31 M. Schoeberl. Time-predictable cache organization. In *STFSSD*, 2009.

32 Freescale Semiconductor. MPC8548E PowerQUICC III integrated processor hardware specifications. http://cache.freescale.com/files/32bit/doc/data_sheet/MPC8548EEC.pdf.

33 SoCLib. The soclib project. http://www.soclib.fr/trac/dev.

34 T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *RTSS*, 2010.

35 STMicroelectronics. STM32F756xx datasheet. http://www.st.com/content/ccc/resource/technical/document/datasheet/fb/d4/56/db/60/61/4f/9c/DM00166114.pdf/files/DM00166114.pdf/jcr:content/translations/en.DM00166114.pdf.

36 Texas Instruments. TMS570LS09x/07x 16/32-Bit RISC flash microcontroller. http://www.ti.com/lit/ug/spnu607/spnu607.pdf.